Lossless Compression: How It Works

BY BARRY SIMON

File compression techniques have been used in utilities such as PKZIP for years. They've also been commonly employed in tape backup systems and, more recently, utilities such as Stacker have made it possible to compress an entire hard disk and perform decompression on the fly.

How do such products accomplish their magic? In this first of a two-part Lab Notes, I'll discuss the bases of the compression systems such as these. In the ne[illegible]sue I'll address the compression methods used to store video and sound information. This division corresponds to the two broad categories into which compression techniques fall: *lossless* and *lossy*.

LOSSLESS VS. LOSSY A lossless system is one that can produce an exact reconstruction of the original file upon decompression. A lossy system—where the compression factor is generally higher—restores only a close approximation of the original.

The appropriate method depends on the type of file you want to compress. A few changed bytes in an executable program will likely cause a crash. If a spreadsheet file is altered by a few bytes, the *best* result you can hope for is that the program will reject the file as corrupt; in the worst-case scenario, the data itself would be changed. And of course it would hardly do if, by compressing and decompressing a memo, you changed the spelling of your boss's name!

On the other hand, given a 24-bit trueco[illegible] file, a few changed pixels wouldn't m[illegible] a noticeable difference. A digital sound file that loses the extreme high frequencies (those above 20,000 Hz) will sound the same to all but your dog. As a rule, files that require lossless compression are composed of *entered data,* while files that permit the use of lossy methods are made up of *sampled data.*

This Lab Notes will provide an overview of lossless compression. (For a detailed explanation of how the compression utility PKZIP works, see the May 25, 1993, Tutor column "Understanding Data Compression" on this topic.) Next time I'll talk about lossy compression methods, and will also describe some of the third-party libraries available to programmers who want to implement compression in their applications.

YOU CAN'T COMPRESS EVERYTHING Not all files can be compressed. Moreover, any lossless algorithm that can decrease the size of some files must *increase* the size of other files. These two basic principles form a good starting point from which to discuss compression.

How do compression utilities shrink files to half their original size and then reliably restore them? The trick is taking advantage of patterns in the files we want to compress.

If *every possible* file could be compressed, then logically it would be possible to successively reduce all files to 0 bytes. Clearly, a 0-byte file cannot contain the information needed to restore the original, nor can it become anything but larger.

Now let's consider files of one byte. Only the 0-byte file is smaller and, as noted, cannot contain the information necessary for decompression. Also, a 0-byte file already represents itself. For lossless compression, the 0-byte file can't represent both the uncompressed 0-byte file and a compressed 1-byte file. If there isn't a unique compressed file for each original file, then the original can't be restored through decompression. So 1-byte files can't be compressed, either.

Suppose we expand the scope of our putative compression algorithm to encompass all the theoretically possible files of 2 bytes or less. If each 1-byte file (and the 0-byte file) must be allowed to retain their size, there aren't any smaller files available into which we could map a 2-byte file. As we consider larger and larger files, the problem remains.

Since 1 byte can have 256 possible values (2^8), there can be 256 unique 1-byte files. Adding these to the one possible 0-byte file yields 257 distinct entities smaller than 2 bytes. So at most, 257 of the 65,536 (2^{16}) possible 2-byte files *could* be compressed to smaller files. But to do this, we would have to map the 1-byte and 0-byte files into the empty 2-byte slots left from the compressed 2-byte files. That is, to reduce the size of some 2-byte files, the 0-byte and 1-byte files must be allowed to grow. In this case we've done a simple swap, but the principle holds true for any compression algorithm: If you don't allow some files to become larger, there will be no smaller files available into which larger files can be mapped.

If you ponder this problem a while, you may come up with a promising solution. Accepting that for some files to shrink other files will have to be allowed to grow in size, you start with an algorithm that does just that. Then you go a step further. If the algorithm shrinks the file, you tack a 0-bit to the start of the compressed file. If the original algorithm makes the file larger, you tack on a 1-bit that is followed by the *original* file, unchanged. Now you have a method that will indeed sometimes increase size, but never by more than one bit!

This trick is useful; indeed, every serious compression method allows the straight *storing* of the original, unchanged file, listing the method as *stored* in the internal directory of the compressed file. But simply storing the files that would otherwise grow doesn't help much, since only a small fraction of all files can be compressed even a little.

We've already seen that only a small fraction of all possible 2-byte files can be compressed, even if we allow the smaller files to grow. There simply aren't enough smaller files. Since there are 256 values for a byte, the number of possible files of *n*+1 bytes is 256 times the number of possible files of *n* bytes. Thus it's never possible to compress more than $\frac{1}{256}$ of a file of a given size even by 1 byte!

We've seen, then, that there is simply no algorithm that can shrink all possible files of a given size. Moreover, only a tiny fraction of all possible files can be shrunk by even 1 byte, and the fraction becomes smaller as the compression factor increases. Fortunately, however, we don't need an algorithm that will compress all theoretically possible files.

A key point that underlies all compression is that the files you and I actually have on our disks are far from random. Most theoretically possible files are files we'd have no interest in at all, so we can safely let these grow and shrink only the files we care about. The secret of compression is to find an algorithm that understands how the files we normally use depart from randomness, and selectively shrinks just those.

NONRANDOM IS BETTER You can prove this to yourself by creating the short Pascal program listed in Figure 1. Running the program will produce a file of 10,000 random bytes. If you put this file through PKZIP, you'll wind up with a file of 10,112 bytes! (The ZIP overhead for a directory entry is 112 bytes.) Plainly, you've achieved no compression.

One element of nonrandomness is the uneven frequency with which various characters occur in a language. For example, in English, the letter *e* is very common, but few words contain the letter *x*. Patterns such as these can be exploited by using variable-length coding, where the length of each character's code depends on the frequency of the character. A parable will illustrate.

The story is told of a fierce tribe of programmers, the Foobari. Their language has four letters: *a, e, i,* and *o*. At the dawn of the computer age, their standards group got together and agreed on a set of two-bit codes for their language. This FSCII code was, in binary,

```
a=00 e=01 i=10 o=11
```

A young entrepreneur, Foo Katz, determined a typical program written in Foobari was 50 percent a's, 25 percent e's, and 12.5 percent each i's and o's. Foo Katz realized if he instead used the codes

```
a=0 e=10 i=110 o=111
```

he could code a typical file in fewer bits. For 25 percent of the time he needed 3 bits instead of FSCII's 2, but 50 percent of the time, he only needed 1. So the average number of bits per character was 1.75 rather than 2—a 12.5 percent decrease in file size—and FKZIP was born.

SHANNON'S THEOREM Could Foo Katz have done better than 1.75 bits per character? Actually, given the probability of each character in Foobari, he couldn't. Work done in the late 1940s by Claude Shannon at Bell Labs shows this to be true. Shannon founded the science of information theory and proved what has come to be called Shannon's theorem.

In its simplest form Shannon's theorem says that if you have *n* possible characters occurring randomly but with nonuniform probabilities p(1), p(2). . .p(*n*), the average number of bits you need to code a character is

$$N = -p(1)\log_2 p(1) - p(2)\log_2 p(2) - \ldots - p(n)\log_2 p(n)$$

In an optimal coding scheme, then, the number of bits needed to code each character is $-\log_2 p(j)$, where $\log_2$ is the logarithm to the base 2. That is, character 1 requires $-\log_2 p(1)$ bits, character 2 requires $-\log_2 p(2)$ bits, and so on. Shannon says you cannot do any better than this. Drawing an analogy with a similar-looking quantity introduced by nineteenth century physicists studying statistical mechanics, Shannon called the quantity N the *entropy*.

As a simple example, suppose all the p(j)'s are equal. If there are *n* characters, the probability of each is p(j) = 1/n. Since $-\log_2(1/n)$ is $\log_2(n)$, we see that

$$N = (p(1) + p(2) + \ldots + p(n))(\log_2(n)) = \log_2(n)$$

This is because, by definition, the sum of all the probabilities of the individual characters must be 1.

The Pascal test program of Figure 1 used characters chosen randomly from all 256 possible values, and we saw that the resulting file could not be compressed at all. However, if you go back to that test program and change random(256) to random(16), the result will be different. When a number smaller than 256 is used,

TESTFILE.PAS

Complete Listing

```
var                                (* variable declarations *)
  i:word;
  b:byte;
randfile:file of byte;

begin                              (* program start *)
  assign(randfile,'foo.bar');      (* give the file the name foo.bar *)
  rewrite(randfile);               (* creates and opens file *)
  for i:=1 to 10000 do begin       (* start loop, execute 10,000 times *)
    b:=random(256);                (* select byte randomly from 256 choices *)
    write(randfile,b);             (* write out the byte *)
  end;                             (* end loop *)
close(randfile);                   (* close file *)
end.                               (* program end *)
```

***Figure 1:* Changing random(256) to random(16) in this program listing turns the resulting file from one that cannot be compressed at all into one that can be compressed by 42 percent.**

the distribution of bytes is not random among all possibilities, even though the byte-to-byte changes are still random. Run it and try PKZIP again—you'll get a 42 percent improvement in file size (using PKZIP 2.0). The theoretical best you can hope for in the random(16) case is 50 percent. This is because only four bits are needed to generate 16 different characters ($N = \log_2(16) = 4$), and four bits are half as much as eight bits.

Now change the program line to random(200). When you run PKZIP again, you'll get a 4 percent improvement over the original. If $n = 200$, then Shannon says each character needs $\log_2(200) = 7.644$ bits, or .9554 bytes. So the theoretical maximum compression savings is 4.5 percent for the random(200) case. This shows that even a slight departure from complete randomness will permit some compression. (Of course, it's not clear how you can *use* fractional bits to represent something, but I'll get to that later.)

For the Foobari example, the p's are ½, ¼, ⅛, and ⅛, and the negative logs are 1, 2, 3, and 3—exactly the number of bits in the FKZIP code that Foo Katz invented. The average number of bits needed is

```
1/2 + (1/4)*2 + (1/8)*3 + (1/8)*3
  = 1.75
```

Thus, the Foo Katz coding did realize the theoretical maximum.

HUFFMAN ENCODING In 1952, Huffman came up with a simple algorithm that realizes the Shannon theoretical maximum when all the probabilities are negative powers of 2 (that is, powers of ½: ½, ¼, ⅛, and so on). Indeed, even when the probabilities are general, the Huffman algorithm works very well. It is also easy to implement in a program. Early versions of ARC, the first widely popular compression scheme on PCs, used Huffman encoding, and it remains one of the elements used in many lossless compression schemes today.

In Huffman encoding, a binary tree is used to assign codes. As depicted in Figure 2, the tree has its root on the left with the branches pointing toward the right. There are some internal *nodes,* which have one line coming from the left and two going off to the right; there are also some *leaves,* which are nodes that have only a line from the left. The leaves are associated with the initial set of characters; there is one leaf for each character.

To form the tree, you first pick the two characters with the smallest probabilities and join them to a node. Give the node the combined probability and then look at that node and the remaining characters. Pick the two objects (nodes or unused characters) with the smallest probabilities, join them to a node, and then continue until all the characters are in the tree.

How the Huffman Algorithm Works

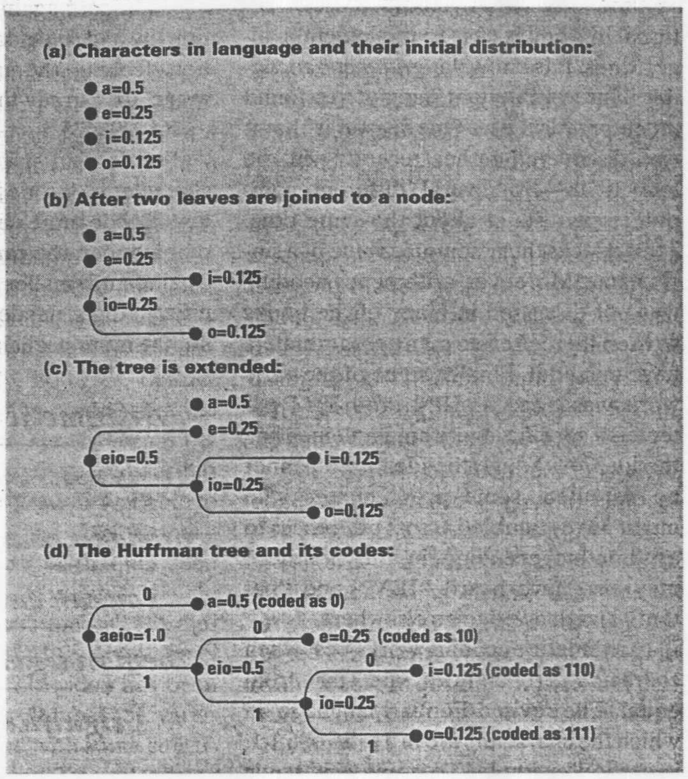

Figure 2: **These tree diagrams show the successive application of the Huffman encoding system to the Foobari language.**

Once you have drawn the tree, you find the code for a leaf by starting at the root and recording the path to that leaf. All leaves on the top branch have a 0 for the first digit; those on the bottom branch have a 1. The number of node levels needed corresponds to the number of bits needed to code the character. Thus the complete tree shown in Figure 2 leads to precisely the coding that Foo Katz used in the Foobari parable.

In the random(200) example, no branch ends before 7 levels, so you will need a minimum of 7 bits to code each character. There are a full 128 nodes or leaves at level 7. Of these 128 items, 56 are leaves and 72 are nodes that each connect to 2 leaves (56 + 2*72 = 200). Thus 144 branches end at level 8.

If every probability is a negative power of 2, the Huffman algorithm perfectly realizes the theoretical maximum predicted by Shannon's theorem and is thus optimal. If some probabilities are not inverse powers of 2, Shannon's maximum won't be realized, but the results are still pretty good. For example, in the random(200) example, Huffman uses an average of 7.72 bits ((56*7 + 144*8)/200), which isn't too much larger than the Shannon limit of 7.644. . .bits.

An alternative tree-based procedure is called Fano-Shannon encoding. Huffman starts with the leaves and combines them, building the tree from the leaves back toward the root. Fano-Shannon uses the opposite strategy: At each stage, it breaks the characters into two groups with roughly equal probabilities and builds the tree from the root toward the leaves. This method produces similar results to Huffman encoding, but some programmers may prefer implementing one over the other.

SAVING UP FRACTIONAL BITS Suppose the Foobari passed a law that henceforth every *e* was to be removed and replaced by an *a*. In that case, text documents would have 12.5 percent *i*'s and *o*'s, but 75 percent *a*'s. Since $-\log_2(.75)$ is 0.415, Shannon tells us that the optimum code length for *a* is less than half a bit. But how in the world can you code a character in less than one bit?

Remarkably, there is an elegant method that both effectively uses fractional bits and is easy to implement in algorithms. It is known as *arithmetic encoding.* Unfortunately, it has not yet found much practical use. One reason is that it was discovered rather recently—in the late 1970s—by several different computer scientists at about the same time. Thus it lacks the mnemonic value of a single name. Moreover, arithmetic encoding was not discussed in many of the books written thereafter, so many programmers have missed it. Finally, some of the early work was done at IBM, and IBM patented a specific algorithm implementing the idea. Although the idea itself cannot be patented, some programmers who might have stumbled across references to arithmethic encoding in the literature may also have heard "IBM" and "patent" and decided to go elsewhere.

Arithmetic encoding works best when character probabilities are far from equal. The revised Foobari language, in which the characters are in a ratio of 6:1:1, is a good example. To apply arithmetic encoding, you should start by thinking of the file to be compressed as a character string—for example, `aio`. You need to associate this string with a real number between 0 and 1. To do this, code the string one character at a time, using the probability of that character to home in on a real number that will represent the string. This real number is not stored as a fixed precision floating-point data type; it's stored as a base 2 number represented as a string of bits.

As we go through an example here, it may help to imagine a ruler whose subdivisions are marked off in sections proportional to the probabilities of all the characters to be used. In Foobari, then, we'd mark off $^3/_4$, and then $^1/_8$ each for *i* and *o*. We'll call the real number that will encode the `aio` string *r*.

Since the first character is *a*, the final code for our string must lie inside the a range, between 0 and 0.75. Now to code the *i*, which follows the *a*, we turn our attention to the a section of the ruler. We take that section, and divide it up proportionally ($^3/_4$, $^1/_8$, $^1/_8$), just as we did when coding the *a*. We've narrowed the range for our final code down to the *i* range: 0.5625 to 0.65625. To code the *o* that follows the *i*, we turn our attention to the *i* section, and again divide it up proportionally and select the o range of this section. Taking the middle value of this range, we can say that `aio` is encoded as r = .650390625.

For longer strings, each successive character is encoded in the same way. As you encode more characters into a single real number, the range you're using gets smaller and smaller so the number of significant digits needed gets larger. Encoding the more probable characters shrinks the range less. For example, encoding an *a* shrinks the range to $^3/_4$ of what it was, but encoding an *i* shrinks the range to $^1/_8$. Arithmetic compression is more effective for probable characters because you can encode more of them in a given number of significant digits.

Arithmetic compression is more effective for probable characters because you can encode more of them in a given number of significant digits.

USING MULTIBYTE BLOCKS Most commercially available compression programs don't limit themselves to a single method of compressing files; they make use of a variety of techniques. All the techniques we've discussed so far are examples of *static, probabilistic encoding* with single bytes. They are probabilistic because they use the likelihood of a character to determine the number of bits needed to code the character. They're static because they assume an a priori set of character probabilities and thus a fixed code. Current lossless compression technology goes beyond this kind of encoding in three ways. It uses

- building blocks of more than one character,
- codes tailored to the file (dynamic encoding), and
- pointers to encode redundant parts of the file (Lempel-Ziv).

To determine how much redundancy of information is in the English language, Shannon asked people to guess words from their first few letters. On this basis (and with some further analysis), computer scientists have come to believe that text files require a certain minimal number of bits per letter. Typical estimates lie in the 1.4- to 1.6-bit range, which may seem rather surprising since there are 26 letters in the alphabet. Still, based on this estimate, text files can't be compressed by more than a factor of 5 or 6 from the usual 8 bits per character.

The fact that text can actually be better compressed depends on the fact that there is much more structure to the English language than just the fact that *e* is more common than *q*. There is a tendency for *t* to be followed by *h* and then by *e*, for example, and a *q* is almost always followed by a *u*. This suggests that instead of using single bytes as the objects to be encoded, it can be useful to deal with multiple-byte blocks.

One way to do that is through *run length encoding* (RLE), which records a character and then the number of times it occurs in a string. RLE works well when there are long strings of a single repeated character, as occurs in some EXE files with preinitialized data, in some graphics files with long runs of pure white or black, and in database files with padded fields. It is rarely the best method for most files, however, because this situation is relatively uncommon. Indeed, Version 2.0 of PKZIP no longer includes an RLE method.

Another approach is a dictionary-based system. You might, for example, build a dictionary composed of the 65,280 most common words and phrases in the English language. (65,280 is 64K—64* 1,024—minus 256; you need to reserve 256 entries for the ASCII codes for letters so you can handle words that are not in the dictionary.) You'd then encode a file in 2-byte pieces (16 bits are needed to address 65,536 different possibilities), where each 2-byte code corresponds to an entry in the dictionary. For words that are not found in the dictionary, this method will double the size of the space that is needed since 2 bytes must be used in order to encode each original byte. On the other hand, the encoding would replace many words of 5 or 6 or even more bytes with codes of only 2 bytes.

e the 64K dictionary possibilities do not occur with equal probability in files to be compressed, you could improve on this scheme by using Huffman or arithmetic encoding to further compress the first iteration of encoded text. In the first pass you'd encode the text by representing it in terms of 2-byte codes, as described above. In the next step, the most common codes would be represented by short bit-strings while the uncommon codes are mapped to longer bit-strings, causing further reduction in size.

This encoding method was used in a

One reason compression works is that ASCII-encoded files have redundant information.

program called Stomp, which was popular on BBSs several years ago. Stomp n caught on because its method is inhe ntly slow and has some additional limitations. It's slow because of the large number of possibilities that must be scanned. It's also limited to text files and requires a large data file (the dictionary) to be on hand. Still, the method is a likely candidate for the tightest compression of text.

DYNAMIC ENCODING Everything I've discussed so far sets up a fixed encoding mechanism for all files. But a set of probabilities that's good for text will be bad for database files, and a set of probabilities that works for database files will be bad for EXE or WK1 files. Clearly, it would be best to tailor the probabilities to the file. That's in reality what every commercial compression package does.

The simplest way to implement file-based encoding is to compute the probability distribution of the bytes used in the file and use those distributions as the basis for Huffman or Fano-Shannon encoding. However, this method requires that the file be read twice. That means it cannot be used for real-time compression ap itions such as modem communications or hard disk backup to tape.

To meet the demands of real-time compression there are methods that recompute probabilities and codes as they go along. The probabilities encountered in the first part of the file are used to optimally encode the later parts. For example, the algorithm might start with an a priori distribution, and then every 256 bytes recompute the distribution based on what has gone before. These methods are called *dynamic* or *adaptive* encoding.

You might think that since the codes keep changing, dynamic encoding systems would present problems for decompression. Fortunately, however, the decompressor can always reconstruct the method the compressor used. This is because at any stage the decompressor has exactly the same information available to it that the compressor had at that stage—namely, the part of the file that has been uncompressed up to that point.

Behind much modern lossless compression is an idea introduced by Lempel and Ziv in 1977–78. This idea has been implemented with a number of variations, especially in one with some contributions by Welch, so you'll see references to the method as LZ or LZW. The basic idea is to use the file itself as a sliding dictionary as it is processed. As the compressing program is about to compress the next *n* bytes of the file, it looks at what it has already processed and sees if the first *x* characters of those bytes have occurred in the past *y* bytes. If so, it stores two pieces of information: a pointer to where the identical string occurred and the number of characters in the string.

For example, the algorithm might look back at the past 1,024 characters for an identical string with a length of 16 or fewer characters. In this example, 10 bits are needed to store the number from 1 to 1,024 that tells how far back the string was, and 4 bits are then needed to store the length of the string. Thus, the total number of bits needed is 14. This contrasts very favorably with the maximum of 128 bits that would be needed to represent a 16-byte string—it's a compression factor of more than 9.

It is a remarkable fact that there are enough repetitions in most files that this method is often very effective. Sometimes supplemented by a probabilistic method, it is the basis of many types of compression programs, including disk compressors (for example, Stacker and DOS 6's DoubleSpace), archivers (for example, PKZIP, ARC, and LHA), and tape backup programs.

USEFUL REDUNDANT INFORMATION One reason compression works is that ASCII-encoded files have redundant information. Once redundancy is removed, however, changing a few bits can change the original file dramatically. For example, changing the ten-bit pointer in the LZ encoding example above could change the decompressed file by 16 bytes! And changing the four-bit length component can effect the meaning of all future pointers and so change the file even more!

For this reason, after squeezing out redundant information, a good compression program puts some redundancy back in the form of a checksum, such as a CRC (cyclic redundancy check). This lets the program warn you if the file has been corrupted in any way. The compression algorithm squeezes out a lot of useless redundancy and then puts back some useful redundancy.

In sum, the two most important techniques in lossless compression are using the nonrandom distribution of bytes within files to determine variable length coding schemes; and using the fact that there are often long repetitions of strings within a file to strip out redundancy. The first is the basis of Huffman and arithmetic coding; the second is the basis of LZ and LZW.

For further reading, I recommend you consider four books. The first two are academic computer science books, and while they are somewhat technical and dry, they also contain a great deal of valuable information. These are Bell, Cleary, and Witten, *Text Compression,* Prentice Hall, 1990, ISBN 0-13-911991-4; and Storer, *Data Compression,* Computer Science Press, 1988, ISBN 0-71678156-5. For a book written more for programmers, see Nelson, *The Data Compression Book,* M&T Press, 1992, ISBN 1-55851-214-4. Finally, for some fascinating reflections on information and data compression, take a look at Lucky, *Silicon Dreams,* St. Martin's Press, 1989, ISBN 0-31202960-8. □

BARRY SIMON IS A CONTRIBUTING EDITOR TO *PC MAGAZINE.*